REPORT DOCUMENTATION PAGE	Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE:	3. REPORT TYPE AND DATES COVERED Final Report 1-Jul-2003 - 31-Aug-2006

4. TITLE AND SUBTITLE Dealing with Suspicious Behavior: Distinguishing Novel Usage from Novel Attacks	5. FUNDING NUMBERS DAAD1903C0060
6. AUTHORS Carla Marceau	8. PERFORMING ORGANIZATION REPORT NUMBER
7. PERFORMING ORGANIZATION NAMES AND ADDRESSES Odyssey Research Associates, Inc. 33 Thornwood Dr., Suite 500 Ithaca, NY 14850 -1250	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSORING / MONITORING AGENCY REPORT NUMBER 43745-CI.2

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12. DISTRIBUTION AVAILIBILITY STATEMENT Approved for Public Release; Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The abstract is below since many authors do not follow the 200 word limit

14. SUBJECT TERMS hybrid intrusion detection system, anomaly and misuse-detection, IDS	15. NUMBER OF PAGES Unknown due to possible attachments
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION ON THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev .2-89)
Prescribed by ANSI Std.
239-18 298-102

Report Title

Distinguishing Novel Usage from Novel Attacks - Final Report

ABSTRACT

In this project, ATC-NY is developing methods for evaluating anomalous behavior concurrently with reacting to it. Anomalous events that are not so suspicious as to cause an immediate alarm are continually reexamined in the light of later events, with the goal of eventually understanding whether they are benign or malign. As time goes on, the IDS should become familiar with common attacks, even while it continually adapts to small changes in normal behavior. By focusing on the long-term problem (building up knowledge), the proposed IDS should become better over time at solving the short-term problem (detecting attacks).

List of papers submitted or published that acknowledge ARO support during this reporting period. List the papers, including journal references, in the following categories:

(a) Papers published in peer-reviewed journals (N/A for none)

"Getting More Out of Your Anomalies: Reasoning about Anomalous Behavior in Systems with Shared Libraries", Eighth Internationa Conference on Information and Communications Security (ICICS '06)

Number of Papers published in peer-reviewed journals: 1.00

(b) Papers published in non-peer-reviewed journals or in conference proceedings (N/A for none)

Number of Papers published in non peer-reviewed journals: 0.00

(c) Presentations

Number of Presentations: 0.00

Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Number of Non Peer-Reviewed Conference Proceeding publications (other than abstracts): 0

Peer-Reviewed Conference Proceeding publications (other than abstracts):

Number of Peer-Reviewed Conference Proceeding publications (other than abstracts): 0

(d) Manuscripts

Number of Manuscripts: 0.00

Number of Inventions:

Graduate Students

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Post Doctorates

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Faculty Supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Under Graduate students supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Personnel receiving masters degrees

NAME

Total Number:

Names of personnel receiving PHDs

NAME

Total Number:

Names of other research staff

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Sub Contractors (DD882)

Inventions (DD882)

ATC-NY TR-06-0009

Distinguishing Novel Usage from Novel Attacks
Final Report

Carla Marceau

September, 2006

Prepared for:
Army Research Organization
U.S. Army Robert Morris Acquisition Center
P.O. Box 12211
Research Triangle Park, North Carolina 27709-2211

Prepared by:
ATC-NY
Cornell Business & Technology Park
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250
Contract No. DAAD19-03-C-0060

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	20-09-2006
2. REPORT TYPE	Final
3. DATES COVERED *(From - To)*	05-06-2003 to 20-09-2006
4. TITLE AND SUBTITLE	Dealing With Suspicious Behavior: Distinguishing Novel Usage From Novel Attacks
5a. CONTRACT NUMBER	DAAD19-03-C-0060
5b. GRANT NUMBER	
5c. PROGRAM ELEMENT NUMBER	F.2
5d. PROJECT NUMBER	
5e. TASK NUMBER	
5f. WORK UNIT NUMBER	
6. AUTHOR(S)	Carla Marceau
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	Odyssey Research Associates, Inc. 33 Thornwood Drive, Suite 500
8. PERFORMING ORGANIZATION REPORT NUMBER	TR06-0009
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	U.S. Army Robert Morris Acquistion Center P.O. Box 12211, Research Triangle Park, NC 27709-2211
10. SPONSOR/MONITOR'S ACRONYM(S)	ARO
11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT	
13. SUPPLEMENTARY NOTES	

14. ABSTRACT

In this project, ATC-NY is developing methods for evaluating anomalous behavior concurrently with reacting to it. Anomalous events that are not so suspicious as to cause an immediate alarm are continually reexamined in the light of later events, with the goal of eventually understanding whether they are benign or malign. As time goes on, the IDS should become familiar with common attacks, even while it continually adapts to small changes in normal behavior. By focusing on the long-term problem (building up knowledge), the proposed IDS should become better over time at solving the short-term problem (detecting attacks).

15. SUBJECT TERMS

hybrid intrusion detection system, anomaly and misuse-detection, IDS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Richard Smith
U	U	U	SAR	13	**19b. TELEPHONE NUMBER** *(Include area code)* (607) 257-1975

Table of Contents

1 Introduction

In this final report, we describe the results of our research on distinguishing novel usage from novel attacks in host-based intrusion detection systems.

1.1 Background

Intrusion detection is vital to the security of mission-critical Army computer systems. It is especially important in today's increasingly wireless environment, which provides many more opportunities for malicious intruders to penetrate Army networks and do a great deal of damage as apparent "insiders."

Today's intrusion detection systems (IDSs) fall into two types. Signature-based IDSs maintain profiles of aberrant behavior and raise an alarm when such behavior occurs; they cannot detect novel attacks. Anomaly-based IDSs maintain profiles of normal behavior and raise an alarm when anything else occurs. However, they often generate false alarms. Like signature-based IDSs, anomaly-based IDSs focus on the problem of the moment: does this behavior indicate an intrusion? In many cases, the honest answer is, "I don't know." The event may be part of an intrusion, but it might simply be rare or novel behavior, perhaps caused by a new software release or a change in communication patterns. Current IDSs either sound the alarm—leading to a high false alarm rate and decreased confidence in the IDS—or quietly forget the possible intrusion.

The focus of this project is improving host-based anomaly detection techniques. The original context of the work was the development of hybrid behavior profiles that include both normal and aberrant behavior. Such profiles would work as shown in Figure 1. Novel behaviors would be cached and information collected about the behavior, so that the IDS could reason about the behavior, eventually resulting in the characterization of a given behavior as either normal or aberrant/malicious.

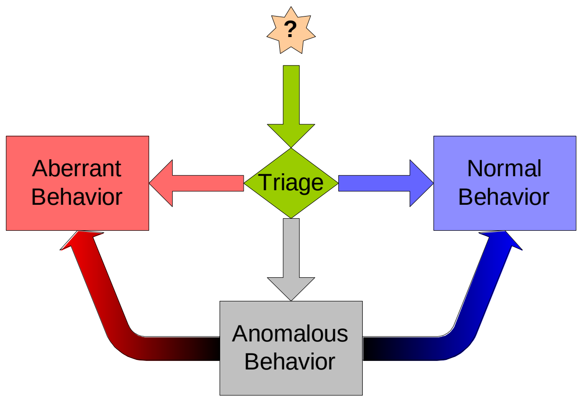


Figure 1. Triage of behaviors

The system that we originally selected as the concrete basis for study is process behavior as characterized by sequences of kernel calls [Forrest96b, Hofmeyr98]. The Forrest group at the University of New Mexico showed that short sequences of calls made by a running program to the operating system kernel can be used to detect attacks. Forrest observed that a program can be characterized by the set of traces of kernel calls that result from executing the program. A priori, it is not obvious how to represent the (usually infinite) set of possible traces, since loops and conditionals can cause each invocation of a program to lead to a different trace. The Forrest group devised a simple but ingenious way to characterize the set of possible traces; they use the set of *N-grams* produced by sliding a window of length N along a trace of a program process. An N-gram is a string of N symbols, each of which corresponds to a kernel call such as *open*. If the program is run many times during a training period, then the result set of N-grams includes most of those that will ever appear in normal traces.

The use of kernel call traces was a major breakthrough in anomaly detection and for many years has provided the best characterization of program behavior. Many alternative detection algorithms have been based on the same data and derive from the N-gram approach, including [Debar98, Ghosh99, Kruegel03, Lee98, Lee99b, Marceau00, Michael02, Sekar01].

1.2 Shared library (DLL) profiles

In the course of this research, while looking for data streams that would support reasoning about anomalies, we discovered a novel way of characterizing program behavior that has numerous advantages over previous approaches. This novel approach provides an alternative to the kernel-call data stream. It is based on tracing calls between independently loaded executables and libraries. Like the work on gray-box anomaly detection [Gao04a, Gao04b] at Carnegie Mellon University, it exploits call and return sequences from the application program through various libraries and functions down to the OS kernel. However, our purpose is to factor profiles and provide a rich new source of data for analyzing anomalies in a hybrid IDS. In addition to supporting reasoning about anomalies, the new method provides a more precise “fit” to actual program behavior, which gives it the potential to reduce false positives (false alarms) and false negatives (missed attacks) simultaneously. The new method, which we call “shared library profiles” or “DLL profiles,” has numerous advantages over kernel-call profiles. It is discussed in the attached paper.

1.3 Summary of progress

In the course of this contract, we have

- Developed a novel family of methods for detecting anomalies in program execution, called “shared library profiling”
- Demonstrated superior intrusion detection qualities of shared library profile techniques
- Demonstrated that the performance penalty of shared library profiling is acceptable
- Developed a novel method of instrumenting applications, called “cascading wrappers,” which is able to capture the data stream for shared library profiling
- Submitted a paper on DLL profiles to ICICS

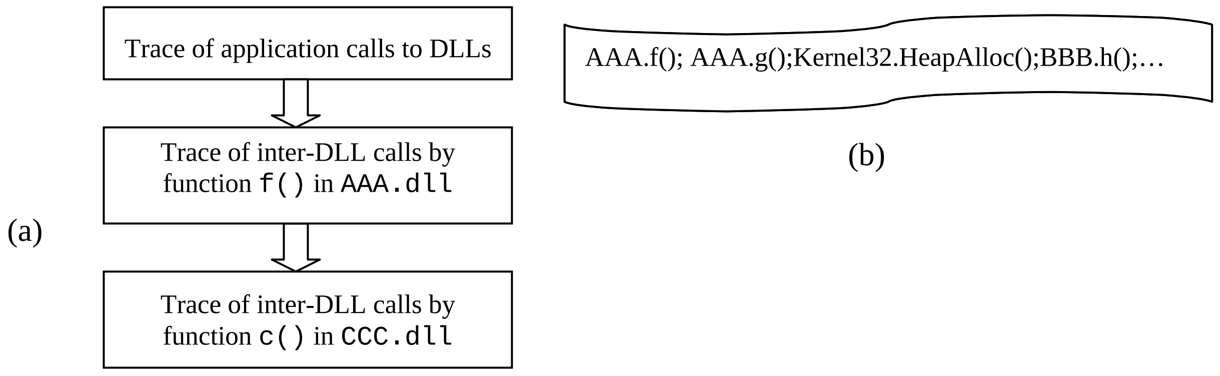
These results are summarized in the following sections.

2 Shared library (DLL) profiles

A Windows process comprises multiple (kernel-supported) threads, some of which are dedicated to GUI or system functions. Windows applications make extensive use of DLLs that implement the operating system and supply additional functionality. The Windows kernel API is defined by ntdll.dll. However, Windows applications rarely call ntdll.dll directly. Indeed, the Microsoft Visual Studio development environment does not support calls to ntdll.dll. Instead, kernel32.dll[1] defines the standard interface to the operating system, although a few DLLs call ntdll directly. Many calls to kernel32 are mediated through higher-level DLLs. As a result, the typical application call causes multiple calls through layers of DLLs that result in possibly many calls to ntdll and the kernel.

Because calls to operations in ntdll typically originate in other Windows DLLs, not in the application, much of the information in kernel-call traces characterizes the internal behavior of other DLLs. Therefore, a single N-gram in such a trace often reflects the behavior of multiple DLLs. In a short execution of Outlook, up to six DLLs at a time were represented on the call stack. Other characterizations of the behavior of the application as a whole also describe the combined behavior of many shared libraries.

In DLL profiling, we characterize each module (the application and the DLLs) by the calls it makes to other DLLs, including ntdll, which implements the kernel API. When one DLL calls another, their combined state can be represented with a *stack* of traces of calls between modules, one for each current invocation of a module. Figure 2 represents a snapshot of the stack. Each box represents a separate sequence that is *currently* being accumulated. In Figure 2, the most recent inter-module call by the application is to function f() in AAA, which in turn has called function c() in CCC. When function c returns, the current inter-DLL sequence for function c() is complete. If function f() calls another function in another DLL, a sequence for that function is pushed onto the stack. Since DLLs are reentrant, the stack may include multiple instantiations of a single module.


Trace of application calls to DLLs
Trace of inter-DLL calls by function f() in AAA.dll
Trace of inter-DLL calls by function c() in CCC.dll
(a)
AAA.f(); AAA.g();Kernel32.HeapAlloc();BBB.h();…
(b)

[1] The name "Kernel32" suggests that this DLL defines an interface to the kernel. Kernel32 provides very basic operating system functionality, but it accesses the kernel only through ntdll, which implements the kernel API. In this report, we will commonly write DLL names without the .dll extension.

Figure 2. (a) The stack of inter-DLL-call sequences in a thread. Each stack element is a trace (b) of calls from an exported function of a DLL (or the application main) to other DLLs.

This characterization includes much the same information as the VtPath model of Feng et al. [Feng03]. Feng et al. exploit the call stack at each system call to record calls and returns between successive system calls. Like the VtPath model, DLL profiles detect anomalies above the kernel-interface level. Our model differs from theirs in that it records the thread history per calling DLL, rather than per call to ntdll. In addition, our model is much sparser because it includes only calls between modules. At any one time, the expected number of functions on the DLL stack is much smaller than the number of functions on the call stack, because functions exported by a DLL are gateways to the DLL's entire functionality, much of which may be implemented in other functions inside that DLL. The exported function may make several calls within the DLL before some function in the DLL makes a call to another DLL.

DLL profiles constitute a whole class of intrusion detection methods, depending on what information is recorded in the traces and the profile for each exported DLL function. For example, if the profile focuses on control flow, training traces record the identity of the called functions. N-grams, automata, or other methods may be used to represent the set of traces [Debar98, Forrest96b, Ghosh99, Hofmeyr98, Marceau00, Pfleger04, Warrender99]. Alternatively, if the profile focuses on dataflow, the training traces record not only the functions called, but also relations among the arguments to the function being profiled and the arguments of the functions it calls. The experiments described in this report used N-grams, with N=6,[2] but most of our results are more generally applicable.

An IDS that uses the DLL stack model for intrusion detection can be realized in a straightforward way. We posit that the IDS maintains a profile of each function exported by a Windows system DLL, in addition to a profile of each application module (binary or DLL) to be protected. At run time, calls to each profiled DLL are captured, for example by mediating connectors [Balzer99a, Balzer99b] or our instrumentation, and sent to the IDS. For each thread, the IDS maintains a stack of currently executing modules (DLLs or the main application). For each function in the stack, it records information about the external calls made by the function, as in Figure 2. When an exported function of a DLL is called from another DLL, the instrumentation informs the IDS of the call. The IDS notes the call in the trace at the top of the DLL stack for that thread, checks for anomalies against the profile of the calling function, and pushes a trace for the called function onto the stack. When the DLL function returns normally, its trace is popped off the DLL stack.

After an update to a DLL, the IDS continues to function but switches to training mode for the updated DLL. When an exported function from the newly updated DLL is called, the IDS pushes the DLL onto the stack, but instead of comparing the trace of the DLL function to the old profile, it collects the trace for input into a new profile. When the DLL function returns, the completed trace is added to the collection of traces for that function, and the profile creation module of the IDS processes it. At some point, the profile is deemed sufficiently mature to be used for detection. At that point, the IDS switches back into detection mode for that DLL

[2] Although Forrest's group used N=6 to model UNIX and Linux processes, a smaller value for N may be more appropriate for tracking behavior in terms of inter-DLL calls.

function. Note that function profiles mature at different rates, depending on how frequently the different functions are exercised.

3 Benefits of shared library (DLL) profiles

In the course of this work, we have identified and provided evidence for five important benefits of DLL profiles. Experimental evidence was based on developing a profile of a small application (ImgViewer32) and using it to exercise exploits against two recently discovered vulnerabilities in Windows DLLs: the gdiplus vulnerability and the WMF vulnerability. Results for both exploits are documented in earlier reports and in a paper.

Localization of anomalous behavior to code modules. In both cases, the vulnerable functions exhibited clearly anomalous behavior. That is not surprising. What may be more surprising is that during both exploits, most of the over 800 DLL functions exhibited completely normal behavior. The profiles of a few functions did not converge during training, so they were ignored when looking for anomalies. Thus, the attacks could easily be associated with the vulnerable modules. This helps in isolating the location of a new vulnerability and understanding which other applications might be vulnerable (those that use the vulnerable DLL).

Reduction of false negatives. The DLL functions we have profiled typically have a narrow range of behavior. 90% of all traces are of length 6 or less, and half call just one other function. This dramatically reduces the chances of a false negative, since it is unlikely that attack behavior happens to fall into the narrow range of the function's normal behavior.

Resistance to mimicry attacks. The narrow range of normal behavior reduces the probability of false negatives and makes mimicry attacks infeasible by making the target much smaller: 2 DLLs instead of 14 and 10 functions instead of over 800. Consider the gdiplus exploit, for example. The exploit payload in our experiment created a new user through calls to the netapi32 DLL. A clever attacker will avoid such blatant behavior, but will find himself constrained by the normal profile of the vulnerable function, in our case GdiplusShutdown. A mimicry attacker has to find a function that is not only vulnerable but also includes the desired functionality. This is much harder to do. For example, GdiplusShutdown does not call any function that creates new users, writes files, or sends messages to another computer.

Anomaly analysis. Localizing anomalies to one or more DLLs makes it possible to draw on knowledge about the DLLs to analyze anomalies. Anomaly analysis in real time helps the IDS decide whether to treat the anomaly as novel application behavior or an attack. For example, the fact that the anomalies caused by the WMF exploit were in code that had been stable for fifteen years made the anomalies much more suspicious than anomalies in a function for which training has barely completed.

Anomaly analysis can also consider the distance of the anomaly from the profile. For example, the gdiplus exploit called two functions in netapi32, which creates new users; netapi32 does not appear in the profile. In addition, anomaly analysis can use information about functions—such as their use of powerful actions, such as writing to files—to estimate their potential harm. Other factors of interest are the provenance and change history of the DLL.

Easing the burden of training (and retraining). The time it takes to create a profile of an application depends on all possible combinations of behaviors of all the DLLs the application invokes. In a DLL profile, variations of behavior in one DLL are confined to the profile for that DLL, avoiding a combinatorial explosion. Further, updates to Windows system DLLs are frequent. When any update affecting an application occurs, a new round of retraining is required. Retraining one DLL is quicker, and allows detection of anomalies in other DLLs to continue while the updated DLL is being trained.

The rate at which profiles converge is apt to vary from one DLL function to another. In our experiments, we were able to detect anomalies in functions whose profiles had converged quickly and unambiguously, while ignoring the behavior of other functions (until their profiles converged).

4 Performance of shared library (DLL) profiles

In addition to the work reported above, we validated that the performance of DLL profiles is adequate. To do this, we instrumented Outlook with wrappers, using techniques described in Section 5. Using the Kernrate Performance monitoring tool [KrView04], we measured the amount of time Outlook spent in both user and privileged modes for four configurations:

Outlook alone. In this configuration, no DLL wrappers were used.

Wrappers but no payload. In this configuration, Outlook DLLs were wrapped, but the wrappers did not do anything.

Wrappers with simple lookup function. This configuration modeled inexpensive intrusion detection—a table lookup of each call made by the function to another DLL. This simple lookup would be appropriate for a DLL function that calls only a small number of functions in other DLLs. It simply ensures that no other functions are called.

Wrappers used in training. This configuration writes information about the call to a log and is analogous to what would be used in training.

We then computed the performance penalty for different combinations of detection functions, including

- The simple lookup function. In our experiments, 90% of all DLL functions made 6 or fewer inter-DLL calls. We let the percentage of simple lookup functions vary from 75% to 90%.
- A training function—for this we used our current training function, which writes to a log file. We surmise that at any one time, training will occur for only a few functions. However, we computed the effect of training percentages between 1 and 16%
- A more complex detection function, whose execution time is halfway between simple lookup and training. We assumed that all DLL functions that were not using simple lookup and were not in training were using this (otherwise unspecified) detection function.

The results of the computation are shown in Figure 3. In every case, the performance penalty is under 5%, which should be acceptable for most applications.

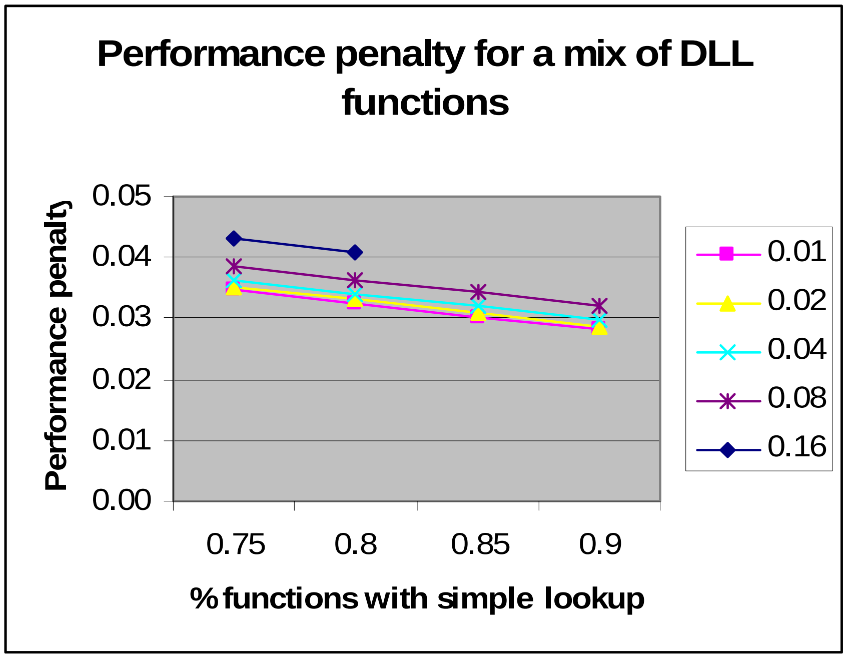


Figure 3. Performance penalty for a mix of IDS techniques and percentage of DLLs requiring training

5 Instrumentation

DLL profiling is of particular interest in Windows systems, which are extremely common. Since we have experience in instrumenting Windows systems, it seemed straightforward to apply our techniques to collecting data on inter-executable calls. In this section, we will describe our previous experience with instrumenting Windows, problems we encountered in our initial attempts to collect data on inter-DLL calls, and the approach that finally brought results.

In another project at ATC-NY, ntdll.dll functions were wrapped so that calls to the kernel could be trapped. In this project, the wrapper technology was further developed to map the locations of DLLs in memory, trace the program stack from the call to ntdll.dll up to the call from the main binary, and to note the transitions from one DLL to another. This technique made it possible to track calls between executables; the technology was also used in a successful DARPA-funded effort [Marceau05] to profile resource use by applications.

However, stack tracing has limitations. For example, many programs are compiled with *stack frame pointer optimization*, which means that they short-circuit the normal calling and register use conventions, making it impossible to identify return pointers on the stack. The basic problem is the lack of a standard stack discipline that makes it possible to reliably trace back from callee to caller on the stack. In order to make the instrumentation more robust, another solution was required. As an alternative, we investigated Microsoft Detours [Detours05], a library for instrumenting Win32 functions by re-writing the target function images, and Teknowledge's mediating connector wrappers [Balzer99]. However, Detours requires identifying every function that is to be instrumented and manually writing "trampoline" instrumentation code for each one, and Balzer's work currently has the same limitation.

We have therefore developed an alternative instrumentation approach, which we call "cascading wrappers." In this approach, whenever a DLL is loaded by the application, the load is intercepted and the DLL wrapped dynamically. (DLL loads are performed by the ntdll.dll function LdrLoadDll.) As a result, all DLLs ultimately invoked by the application are wrapped and instrumented, and invocations of functions within them are logged. To catch invocations to other DLLs through a function pointer, we trap the caller's request for the function pointer and return a pointer to the (wrapped) DLL function.

We are not aware of previous efforts to do automatic dynamic wrapping. Both Teknowledge's wrappers and Microsoft's Detours product require prior knowledge of the API and custom programming to capture calls. Because it is automatic, our technique cannot exploit knowledge of the API or of argument values; we are, however, are able to log the fact that a call has been made.

Both our wrappers and Teknowledge wrappers have "blind spots"—that is, neither can detect all calls made between Windows DLLs, although both can detect most calls. Teknowledge wrappers are defined on exported functions, while ours are based on imported functions. However, two types of function evade detection by either method, because they invoke functions that do not appear in the DLL's export table:

- If a DLL implements a C++ class, a non-exported function may be called through a C++ class table. These calls are invisible to both Teknowledge wrappers and ours.
- In certain other cases, the caller has independent knowledge of the structure of the called DLL and is able to locate a function in the DLL without using the export table.

We believe that additional research on DLL instrumentation would be of great benefit to advancing the state of the art in host-based intrusion detection.

6 Publications

C. Marceau and M. Stillerman, "Modular Behavior Profiles in Systems with Shared Libraries," submitted to the *Eighth International Conference on Information and Communications Security (ICICS '06),* 2006.

7 Conclusions

In the past three years, we have implemented and tested a novel data stream for host-based anomaly detection that helps to distinguish between novel behavior and novel attacks. This data stream results in a closer approximation to actual program behavior than has hitherto been available and makes it possible to reduce both false positives and false negatives, discourage mimicry attacks, reduce the burden of training, and provide information for anomaly analysis. DLL profiles lead to a family of intrusion detection methods that all enjoy these advantages.

8 References

[Balzer99] R. Balzer and N. Goldman, "Mediating Connectors," in *Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshop*, 1999, pp. 73-77.

[Balzer99a] R. Balzer and N. Goldman, "Mediating Connectors," in *Proceedings of ICDCS Workshop on Electronic Commerce and Web-Based Applications*, 1999, pp. 73-77.

[Balzer99b] R. Balzer and N. Goldman, "Mediating Connectors: A Non-Bypassable Process Wrapping Technology," in *Proceedings of 19th IEEE International Conference on Distributed Computing Systems*, 1999.

[Debar98] H. Debar, M. Dacier, M. Nassehi and A. Wespi, "Fixed vs. Variable-Length Patterns for Detecting Suspicious Process Behavior," in *Proceedings of ESORICS 98, 5th European Symposium on Research in Computer Security*, 1998.

[Detours05] Microsoft Research, "Detours," http://research.microsoft.com/sn/detours/.

[Feng03] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of IEEE Security and Privacy*, 2003.

[Forrest96b] S. Forrest, S. A. Hofmeyr and A. Somajayi, "A Sense of Self for UNIX Processes," in *Proceedings of IEEE Symposium on Computer Security and Privacy*, 1996.

[Gao04a] D. Gao, M. K. Reiter and D. Song, "On Gray-Box Program Tracking for Anomaly Detection," in *Proceedings of 13th USENIX Security Symposium*, 2004, pp. 103-118.

[Gao04b] D. Gao, M. K. Reiter and D. Song, "Gray-Box Extraction of Execution Graphs for Anomaly Detection," in *Proceedings of 11th ACM Conference on Computer and Communications Security*, 2004, pp. 318-329.

[Ghosh99] A. K. Ghosh, A. Schwatzbard and M. Shatz, "Learning Program Behavior Profiles for Intrusion Detection," in *Proceedings of 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, 1999.

[Hofmeyr98] S. A. Hofmeyr, S. Forrest and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," in *Journal of Computer Security* 6 (3), pp. 151–180, 1998.

[Kruegel03] C. Kruegel, D. Mutz, W. Robertson and F. Valeur, "Bayesian Event Classification for Intrusion Detection," in *Proceedings of 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 2003.

[KrView04] Microsoft (TM), "Krview - the Kernrate Viewer," http://www.microsoft.com/whdc/system/sysperf/krview.mspx.

[Lee98] W. Lee and S. J. Stolfo, "Data Mining Approaches for Intrusion Detection," in *Proceedings of 7th USENIX Security Symposium*, 1998.

[Lee99b] W. Lee, S. J. Stolfo and K. Mok, "A Data Mining Framework for Building Intrusion Detection Models," in *Proceedings of IEEE Symposium on Security and Privacy*, 1999.

[Marceau00] C. Marceau, "Characterizing the Behavior of a Program Using Multiple-Length N-Grams," in *Proceedings of New Security Paradigms Workshop*, 2000.

[Marceau05] C. Marceau and R. Joyce, "Empirical Privilege Profiling," in *Proceedings of New Security Paradigms Workshop*, 2005.

[Michael02] C. C. Michael and A. Ghosh, "Simple, State-Based Approaches to Program-Based Intrusion Detection," in *ACM Transactions on Information and System Security* 5 (3), pp. 203-237, 2002.

[Pfleger04] K. Pfleger, "On-Line Cumulative Learning of Hierarchical Sparse N-Grams," in *Proceedings of International Conference on Development and Learning,* 2004.

[Sekar01] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proceedings of IEEE Symposium on Security and Privacy*, 2001, pp. 144-155.

[Warrender99] C. Warrender, S. Forrest and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of IEEE Symposium on Security and Privacy*, 1999, pp. 133-145.